Explorando o Paralelismo de Stream em CPU e de Dados em GPU na Aplicação de Filtro Sobel

Charles Michael Stein, Dalvan Griebler

[1] Laboratório de Pesquisas Avançadas para Computação em Nuvem (LARCC)
Faculdade Três de Maio (SETREM) – Três de Maio – RS – Brasil

charlesmst@gmail.com,dalvan.griebler@acad.pucrs.br

Resumo. *O objetivo deste estudo é a paralelização combinada do stream em CPU e dos dados em GPU usando uma aplicação de filtro sobel. Foi realizada uma avaliação do desempenho de OpenCL, OpenACC e CUDA com o algorítimo de multiplicação de matrizes para escolha da ferramenta a ser usada com a SPar. Concluiu-se que apesar da GPU apresentar um speedup de 11.81x com CUDA, o uso exclusivo da CPU com a SPar é mais vantajoso nesta aplicação.*

1. Introdução

As aplicações de processamento de *Stream* estão presentes em diversas áreas. Por exemplo, no monitoramento de eventos sísmicos, nas análises de mercado de bolsa de valores, tratamento de imagem, áudio e vídeo. Uma das características destas aplicações é o processamento de um fluxo contínuo de dados e a estrutura bem definida em uma sequencia de operações, com o formato de uma linha de produção. Muitas destas aplicações possuem requisitos de desempenho associados ao processamento de tempo real e a alta vazão. Assim, torna-se necessário a introdução do paralelismo a fim de atender estas demandas.

Existem diversas bibliotecas ou *frameworks* de programação paralela para diferentes arquiteturas atualmente disponíveis. Somado a isso, existem também padrões paralelos que auxiliam o programador a decompor um problema [McCool et al. 2012]. Para explorar o paralelismo de *stream*, os mais usados são os padrões *Farm* e *Pipeline*, que são muitas vezes combinados [Griebler and Fernandes 2013]. No paralelismo de dados, os mais usados são o *Map* e o *Reduce*, que podem ser combinados [Danelutto et al. 2017]. Além disso, é conhecido que problemas que podem ser decompostos independentemente sobre um conjunto de dados são mais recomendados para GPU, pois trata-se de uma arquitetura criada para este cenário. Enquanto isso, a CPU possui uma unidade de controle mais robusta e suporta eficientemente o paralelismo de dados, tarefas e *stream*.

Desta forma, o objetivo deste artigo é realizar a exploração do paralelismo nestes dois níveis arquiteturais em uma aplicação de filtro sobel. A proposta inicial é avaliar quais das interfaces de programação paralela (OpenCL, OpenACC e CUDA) oferecem o melhor desempenho através da paralelização do algorítimo de multiplicação de matrizes. Uma vez definida a melhor, o problema de pesquisa é descobrir se o desempenho nesta aplicação aumenta ou diminui fazendo o uso combinado do paralelismo de *stream* na CPU com a SPar e paralelismo de dados na GPU com a interface escolhida. O desempenho desta aplicação com a SPar [Griebler et al. 2017] pode ser observado em trabalho anteriores, aplicando diferentes estratégias [Griebler et al. 2015]. Consequentemente, a principal contribuição está no suporte ao paralelismo para GPU e sua combinado nesta aplicação de filtro sobel. O artigo apresenta primeiramente as avaliações e o desenvolvimento na Seção 2 e depois discute os resultados dos experimentos realizados na Seção 3.

2. Avaliação e Desenvolvimento

Primeiramente, iremos fazer uma avaliação do desempenho das interfaces de programação paralelas mais utilizadas para explorar o paralelismo nas GPUs. Diante disso, foram implementadas diversas versões do algoritmo de multiplicação de matrizes com CUDA, OpenCL e OpenACC. Dentre estas, foram escolhidas as versões que ofereceram o melhor desempenho para plotar no gráfico da Figura 1. Estas versões foram desenvolvidas da seguinte forma. **CUDA**: Dentro do *kernel*, a multiplicação é feita em blocos de 8x8. As *threads* copiam de forma conjunta matrizes 8x8 para a memória compartilhada e fazem o cálculo de forma cumulativa. **OpenCL**: Semelhante à versão em CUDA, cada grupo de trabalho faz cache de matrizes 8x8, e os *threads* realizam o cálculo de forma cumulativa. **OpenACC**: A implementação possui anotações nos dois laços mais externos e a soma dos valores utilizou a redução do OpenACC. As transferências de memória foram feitas de forma explicita.

Como pode ser visto, o CUDA foi a biblioteca que conseguiu obter o melhor desempenho, alcançando 36x de *speedup* em matrizes 2000x2000. Para estes testes, foi utilizado o mesmo *hardware* e configuração descrito na Seção 3. O melhor desempenho do CUDA ocorre devido ao OpenACC e OpenCL suportarem outras arquiteturas, gerando assim uma implementação mais genérica.

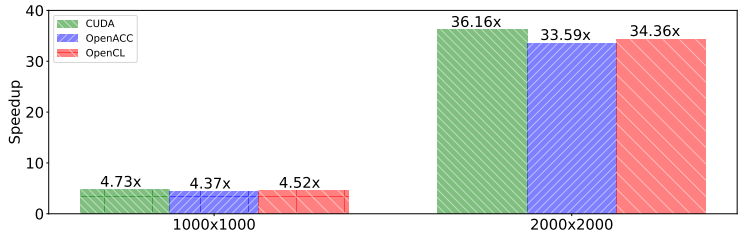

Figura 1. *Speedup* das versões otimizadas para multiplicação de matrizes.

A aplicação de filtro sobel le todas as imagens de um diretório e destaca as bordas das imagens. O operador sobel é aplicado em cada pixel da imagem, que leva em consideração os pixeis vizinhos para obter um valor que representa a diferença de corres. O Código 1 apresenta a implementação desta aplicação usando a SPar. Para cada arquivo, um trabalhador lê, aplica o filtro e salva o resultado.

O Código 2 apresenta o operador/função sobel em formato de um kernel CUDA. Como pode ser observado, a execução ocorre em blocos que são calculados baseado no tamanho da imagem. O *kernel* aplica o filtro no índice atual da imagem e escreve o resultado em uma variável global, que ao final é transferida para a CPU. Note que ambos os códigos são versões totalmente voltados para a sua arquitetura alvo. Para realizar a combinação, foi mantida as anotações do Código 1, porém, ao invés de chamar o operador sobel sequencial na linha 10, inseriu-se todo o código necessário para inicializar e chamar o *kernel* do Código 2, usando a biblioteca CUDA.

A transferência de dados entre CPU e GPU foi implementada de três formas diferentes (resultados estão na Figura 3). A primeira faz a gestão do dados de forma explicita com a API do CUDA. A outra utilizou o recurso *stream*, permitindo que a comunicação CPU e GPU e o processamento dos *kernels* ocorram de forma simultânea. Por último, foi utilizado um recurso *Unified Memory* que faz as transferências automaticamente.

```
DIR *dptr = opendir(...);
struct dirent *dfptr;
[[spar::ToStream, spar::Input(dptr, dfptr,
    tot_img, tot_not), spar::Output(tot_img,
     tot_not)]]
while ((dfptr = readdir(dptr)) != NULL){
 // preprocessing
 if (file_extension == "bmp"){
  tot_img++;
  im = read(name, h, w);
  [[spar::Stage, spar::Input(h, w, im,
    newname), spar::Output(new_im), spar::
    Replicate(2)]]{
   new_im = sobel(im, h, w);
   write(newname, new_im, h, w);
  } // end stage
 }else{ tot_not++; }
}
```

Código 1. Aplicação com SPar.

```
__global__ void sobel_k(unsigned char *fi,
    unsigned char *im, int w, int h){
    int y = blockIdx.y * blockDim.y +
    threadIdx.y + 1;
    int x = blockIdx.x * blockDim.x +
    threadIdx.x + 1;
    unsigned char b[3][3];
    if (x < (w-1) && y < (h-1)){
        for (int v = 0; v < 3; v++){
            for (int u = 0; u < 3; u++){
                b[v][u] = im[(((y + v - 1) *
     w) + (x + u - 1))];
                fi[((y*w)+x)]=Sobel(b);
            }
        }
    }
}
```

Código 2. Sobel em CUDA

3. Experimentos

Para analisar o desempenho das versões implementadas, foram usadas sempre 100 imagens com dimensões de 800x600, 3000x2250, 5000x5000 e 10000x10000. O *hardware* utilizado foi um computador com CPU Intel Xeon E5-2620 v3 @2.40GHz de 12 núcleos, possuindo uma GPU Titan X(pascal) de 12GB e 3584 CUDA cores.

O primeiro teste foi a comparação do desempenho da implementação CUDA com a sequencial. A Figura 2 apresenta o desempenho nas diferentes cargas de dados, destacando o tempo total da execução e o tempo da região paralela (operador sobel em CUDA). Como pode ser visto na Figura 2, a região paralela obteve ganho de desempenho a partir de imagens 3000x2250, com um *speedup* de 1.86x a 11.81x, porém, isso não refletiu em um ganho de desempenho no tempo total de mesma escala. O gargalo na leitura e escrita de arquivos não permite a utilização eficiente da GPU.

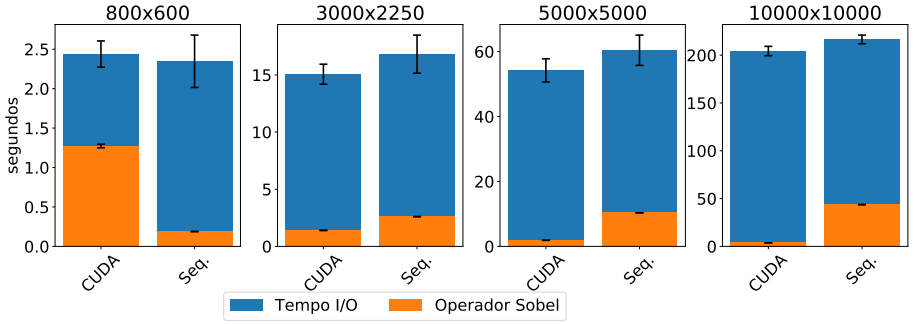

Figura 2. CUDA vs sequencial

A SPar permitiu o paralelismo de *Stream* para CPU de forma simples e ao mesmo tempo eficiente conforme podemos ver na Figura 3, onde o tempo total de processamento das diferentes versões desenvolvidas foi plotado. Nota-se que a execução em imagens de baixa resolução na GPU apresentou menor desempenho que as versões paralelizadas apenas em CPU (versão SPar). Em imagens a partir de 5000x5000, o desempenho entre versões que utilizam GPU se assemelha ao uso exclusivo de CPU. Embora existe o suporte ao paralelismo nos dois níveis arquiteturais, a frequente cópia dos dados e as operações de

I/O afetaram a escalabilidade nesta aplicação. Na carga de imagens 5000x5000, nota-se claramente um problema de balanceamento de carga com a SPar, ocorrendo também nas outras cargas quando testado com um número de réplicas em específico. O desempenho poderia ser melhorado usando o escalonador sob-demanda da SPar (`spar_ondemand`).

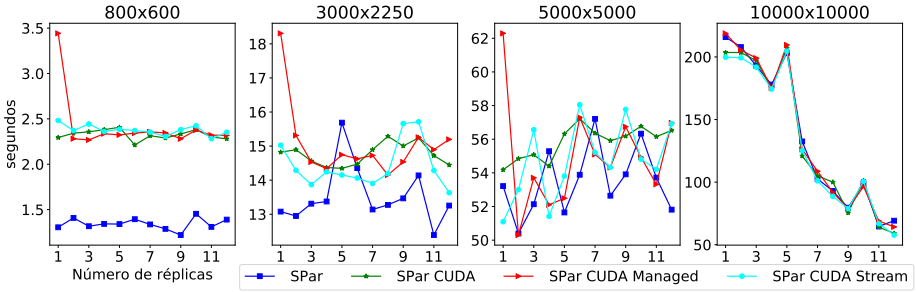

Figura 3. Resultados com o uso combinado do paralelismo em CPU e GPU.

4. Conclusões

Este artigo fez uma análise da aplicação de filtro sobel para GPU em CUDA e também o seu uso combinado com o paralelismo da CPU com a SPar. Apesar do operador sobel apresentar um *speedup* de até 11.81x em CUDA, este desempenho não refletiu em ganhos no tempo total, já que o grande limitador da aplicação foi o tempo de I/O. Neste sentido, observou-se que é melhor utilizar apenas o paralelismo em CPU, já que o processamento realizado na GPU não justifica o *overhead* da transferência de dados. Como trabalhos futuros, almeja-se avaliar se outras aplicações de *stream* conseguem aumentar o desempenho explorando o paralelismo combinado de CPU e GPU.

Referências

[Danelutto et al. 2017] Danelutto, M., de Matteis, T., de Sensi, D., Mencagli, G., and Torquati, M. (2017). P3ARSEC: Towards Parallel Patterns Benchmarking. In *32nd Annual ACM Symposium on Applied Computing*, SAC '17, Marrakech, Morocco. ACM.

[Griebler et al. 2015] Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. (2015). An Embedded C++ Domain-Specific Language for Stream Parallelism. In *International Conference on Parallel Computing*, ParCo'15, pages 317–326, Edinburgh, UK. IOS Press.

[Griebler et al. 2017] Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. (2017). SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):20.

[Griebler and Fernandes 2013] Griebler, D. and Fernandes, L. G. (2013). Towards a Domain-Specific Language for Patterns-Oriented Parallel Programming. In *Programming Languages - 17th Brazilian Symposium - SBLP*, volume 8129 of *Lecture Notes in Computer Science*, pages 105–119, Brasilia, Brazil. Springer Berlin Heidelberg.

[McCool et al. 2012] McCool, M., Robison, A. D., and Reinders, J. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. MKF, MA, USA.